Understanding the 0CFA Abstraction

Kristopher Micinski
CIS 700 —Program Analysis: Foundations and Applications
Fall ’19, Syracuse University

AAM is a **general** strategy for building abstract interpreters from abstract machines

But what do the results **mean**?

For the first part of this lecture—let’s just figure out the answer intuitively (without implementing the analysis)

This week we will study **finite** analyses

Today, will study 0CFA—finite flow analysis for Scheme

On Wednesday, k-CFA—Improves precision of 0CFA

0CFA

Asks the question:
for each **variable**, which possible **lambdas** could be **bound** to that variable?

```
((lambda (x) x) (lambda (y) y))
```

Nothing..

0CFA

Asks the question:
for each **variable**, which possible **lambdas** could be **bound** to that variable?

```
((lambda (x) x) (lambda (y) y))
```

Nothing..

Also…
`(lambda (y) y)` flows to result of **entire application**…

The issue is that this could require transitive reasoning…

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

Rules for 0CFA

For the lambda calculus…

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

- For each lambda term, (lambda (x) e):
  - `(lambda (x) e)` flows to itself
- For each application `(e0 e1)`…
  - If `(lambda (x) e’)` flows to `e0` and v flows to `e1`…
  - Then v flows to x
- For each application `(e0 e1)`:
  - If `(lambda (x) e)` flows to `e0` and v flows to body e…
  - Then v flows to `(e0 e1)`

Rules for 0CFA

For the lambda calculus…

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

- For each lambda term, (lambda (x) e): **Base Case**
  - `(lambda (x) e)` flows to itself
- For each application `(e0 e1)`… **Calls**
  - If `(lambda (x) e’)` flows to `e0` and v flows to `e1`…
  - Then v flows to x
- For each application `(e0 e1)`: **Returns**
  - If `(lambda (x) e)` flows to `e0` and v flows to body e…
  - Then v flows to `(e0 e1)`

- For each lambda term, (lambda (x) e):
  - `(lambda (x) e)` flows to itself
- For each application `(e0 e1)`…
  - If `(lambda (x) e’)` flows to `e0` and v flows to `e1`…
  - Then v flows to x
- For each application `(e0 e1)`:
  - If `(lambda (x) e)` flows to `e0` and v flows to body `e`…
  - Then v flows to `(e0 e1)`

0CFA assigns a set of expressions to each bound variable **and** each subexpression in the program

Intuitively, “what flows to each variable and returns to each control point.”

`((lambda (x) x) (lambda (y) y))`

- For each lambda term, (lambda (x) e):
  - `(lambda (x) e)` flows to itself
- For each application `(e0 e1)`…
  - If `(lambda (x) e’)` flows to `e0` and v flows to `e1`…
  - Then v flows to x
- For each application `(e0 e1)`:
  - If `(lambda (x) e)` flows to `e0` and v flows to body e…
  - Then v flows to `(e0 e1)`

- (lambda (x) x) flows to itself, (lambda (y) y) flows to itself
- For the application:
  - (lambda (x) x) flows to (lambda (x) x) and…
  - (lambda (y) y) flows to (lambda (y) y)
  - So (lambda (y) y) flows to x
- (Now, the third rule…)
  - We just decided that (lambda (y) y) flows to x
  - Thus, (lambda (y) y) also flows to the entire application!

`(((lambda (x) x) (lambda (y) y)) (lambda (z) z))`

- For each lambda term, (lambda (x) e):
  - `(lambda (x) e)` flows to itself
- For each application `(e0 e1)`…
  - If `(lambda (x) e’)` flows to `e0` and v flows to `e1`…
  - Then v flows to x
- For each application `(e0 e1)`:
  - If `(lambda (x) e)` flows to `e0` and v flows to body e…
  - Then v flows to `(e0 e1)`

Work through this one on board…

Now try Ω…

- For each lambda term, (lambda (x) e):
  - `(lambda (x) e)` flows to itself
- For each application `(e0 e1)`…
  - If `(lambda (x) e')` flows to `e0` and v flows to `e1`…
  - Then v flows to x
- For each application `(e0 e1)`:
  - If `(lambda (x) e)` flows to `e0` and v flows to body `e`…
  - Then v flows to `(e0 e1)`

```
((lambda (x) (x x)) (lambda (y) (y y)))
```

We can intuitively extend the lambda calculus with various constructs fairly easily…

```
((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z)))
```

```
(((lambda (x) x)
  (if #t (lambda (y) y) (lambda (z) z)))
 (lambda (a) a))
```

Practice: what flows where?

Data flow depends on **control** flow

```
((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z)))
```

```
(((lambda (x) x)
  (if #t (lambda (y) y) (lambda (z) z)))
 (lambda (a) a))
```

Note: also need to know what expressions flow to which control points

```
((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z)))
```

```
(((lambda (x) x)
  (if #t (lambda (y) y) (lambda (z) z)))
 (lambda (a) a))
```

One choice: anything that flows to either side of an if could return from the entire if

(Can refine via adding precision, will do this next.)

This is an approximation, because at runtime **lambdas** don't get bound to variables, but **closures** do.

0CFA conflates all possible environments that could be closed alongside a piece of syntax

0CFA "ignores" the environment

0CFA

```
((lambda (x) x) (lambda (y) y))
```

$\rho = []$

Closure

Can treat let/let* similarly to lambdas…

```
(let* ([id (lambda (x) x)]
       [foo (id 5)])
  (id 3))
```

What does 0CFA say x will get bound to..?

Can treat let/let* similarly to lambdas…

```
(let* ([id (lambda (x) x)]
       [foo (id 5)])
  (id 3))
```

Subsequently, 0CFA says that any return from id also returns to any callsite for id

Note: this illustrates ambiguity in 0CFA!

Thinking back to last time…

$\hat{\varsigma} \longmapsto_{\widehat{CESK}^{\star}_{t}} \hat{\varsigma}'$, where $\kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$

$\langle x, \rho, \hat{\sigma}, a, t \rangle$	$\langle v, \rho', \hat{\sigma}, a, u \rangle$ where $(v, \rho') \in \hat{\sigma}(\rho(x))$
$\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \hat{\sigma}, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle$

Figure 5. The abstract time-stamped CESK* machine.

[Van Horn and Might, ’10]

Defines a **family** of interpreters
(Instantiate by choosing alloc / tick appropriately.)

$$\hat{\varsigma} \longmapsto_{\widehat{CESK}^\star_t} \hat{\varsigma}', \text{ where } \kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$$

$$\begin{array}{l|r}
\langle x, \rho, \hat{\sigma}, a, t \rangle & \langle v, \rho', \hat{\sigma}, a, u \rangle \text{ where } (v, \rho') \in \hat{\sigma}(\rho(x)) \\
\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle & \langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle \\
\langle v, \rho, \hat{\sigma}, a, t \rangle & \\
\text{if } \kappa = \mathbf{ar}(e, \rho', c) & \langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle \\
\text{if } \kappa = \mathbf{fn}((\lambda x.e), \rho', c) & \langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle
\end{array}$$

Figure 5. The abstract time-stamped CESK* machine.

[Van Horn and Might, ’10]